King Mongkut's University of Technology Thonburi

Priority Queues

CSC 209 Data Structures

Dr. Kulwadee Somboonviwat

School of Information Technology, KMUTT

kulwadee.som [at] sit.kmutt.ac.th

Lecture Plan

- Priority Queues API
- Elementary Implementations
- Binary Heap Data Structure
- Heap based Priority Queues
- Heapsort

Abstract Data Types

- A **data type** is a set of values and a set of operations on those values.
- **Primitive types**
  - values immediately map to machine representations
  - Operations immediately map to machine instructions
  - **Java primitive data types:** boolean, byte, char, short, int, long, float, double

Java primitive data types

Type	Description	Default	Size	Examples of Operations
boolean	true or false	false	1 bit	&, !, |, ^
byte	twos complement integer	0	8 bits	+, -, *, /
char	Unicode character	`\u0000`	16 bits	compare
short	twos complement integer	0	16 bits	+, -, *, /
int	twos complement integer	0	32 bits	+, -, *, /
long	twos complement integer	0	64 bits	+, -, *, /
float	IEEE 754 floating point	0.0	32 bits	+, -, *, /
double	IEEE 754 floating point	0.0	64 bits	+, -, *, /

Abstract Data Types

- A **data type** is a set of values and a set of operations on those values.
- **Primitive types**
  - values immediately map to machine representations
  - Operations immediately map to machine instructions
  - **Java primitive data types:** boolean, byte, char, short, int, long, float, double
- In practice, we want to write programs that process data types specific to our applications such as Color, Picture, Complex numbers, Vectors, …

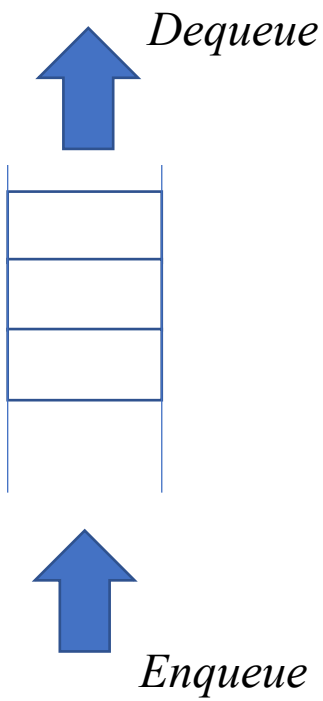
- An **abstract data type** is a data type whose representation is hidden from the client

Collections

- A **collection** is a data type that stores a group of items.

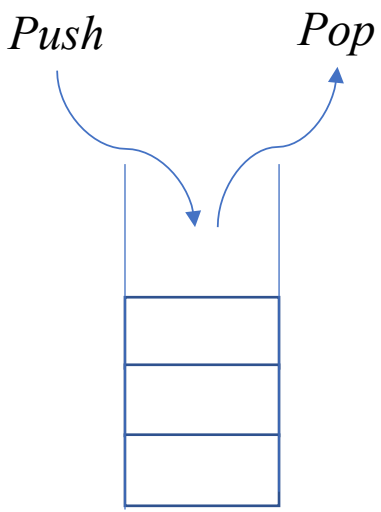
Abstract data type	Core operation	Data structures
Stack	Push, Pop	Array, Linked List
Queue	Enqueue, Dequeue	Array, Linked List
Set	Add, Contains, Delete	Binary search tree, hash table
Symbol table	Put, Get, Delete	Binary search tree, hash table
Priority Queue	**Insert, delete-max**	**Unordered array, Ordered array, Binary Heap**

QUEUE (FIFO)



Remove the item ***least recently added***

STACK (LIFO)



Remove the item ***most recently added***

PRIORITY QUEUE

operation	*argument*	*return value*
insert	P	
insert	Q	
insert	E	
remove max		Q
insert	X	
insert	A	
insert	M	
remove max		X
insert	P	
insert	L	
insert	E	
remove max		P

Remove the ***largest*** *(or* ***smallest****) item*

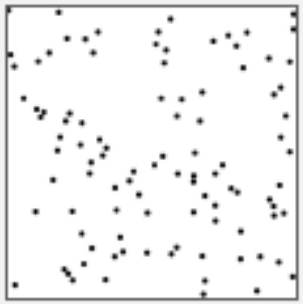
Priority Queue API

public class MaxPQ<Key>		
	MaxPQ()	*create an empty priority queue*
	MaxPQ(Key[] a)	*create a priority queue with given keys*
void	insert(Key v)	*insert a key into the priority queue*
Key	delMax()	*return and remove a largest key*
boolean	isEmpty()	*is the priority queue empty?*
Key	max()	*return a largest key*
int	size()	*number of entries in the priority queue*

```
// find top-m transactions with smallest values
MaxPQ<Transaction> pq = new MaxPQ<Transaction>();
while (StdIn.hasNextLine()) {
    String line = StdIn.readLine();
    Transaction transaction = new Transaction(line);
    pq.insert(transaction);
    if (pq.size() > m)
        pq.delMax();
}
```

Priority queue: applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Discrete optimization. [bin packing, scheduling]
- Artificial intelligence. [A* search]
- Computer networks. [web cache]
- Operating systems. [load balancing, interrupt handling]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Number theory. [sum of powers]
- Spam filtering. [Bayesian spam filter]
- Statistics. [online median in data stream]



Lecture Plan

- Priority Queues API
- Elementary Implementations
- Binary Heap Data Structure
- Heap based Priority Queues
- Heapsort

Using Unordered / Ordered Arrays

operation	*argument*	*return value*	*size*	*contents (unordered)*	*contents (ordered)*
insert	P		1	P	P
insert	Q		2	P Q	P Q
insert	E		3	P Q E	E P Q
remove max		Q	2	P E	E P
insert	X		3	P E X	E P X
insert	A		4	P E X A	A E P X
insert	M		5	P E X A M	A E M P X
remove max		X	4	P E M A	A E M P
insert	P		5	P E M A P	A E M P P
insert	L		6	P E M A P L	A E L M P P
insert	E		7	P E M A P L E	A E E L M P P
remove max		P	6	E M A P L E	A E E L M P

A sequence of operations on a priority queue

```
import java.util.NoSuchElementException;
public class MaxPQStr {
  private String[] pq;
  private int n;
  private static int initCapacity = 20;
  public MaxPQStr() {
    pq = new String[initCapacity];
    n = 0;
  }
  public void insert(String key) {
    if (n == pq.length-1) resize(2*pq.length);
    pq[++n] = key;
  }
  public String delMax() {
    if (isEmpty()) throw new NoSuchElementException("Priority queue underflow");
    // find maximum item
    int maxidx = 1;
    for (int i = 2; i <= n; i++)
      if (less(maxidx, i)) maxidx = i;
    String maxitem = pq[maxidx];
    if (maxidx < n)
      exch(maxidx, n);
    n = n - 1;
    if ((n == 0) && (n == (pq.length - 1)/4)) resize(pq.length/2);
    return maxitem;
  }
  public boolean isEmpty() {
    return n==0;
  }
  public int size() {
    return n;
  }
```

Unordered array

This implementation applies to String items only.

How to make it applicable to *any* data type?

- Solution: Generics

Unordered list with Generics

```
import java.util.NoSuchElementException;
public class LLMaxPQ<Key> {
  private Node<Key> first, last;
  private int n;
  private static class Node<Key> {
    private Key key;
    private Node<Key> next;
  }
  public LLMaxPQ() {
    first = null; last = null; n = 0;
  }
  public void insert(Key k) {
    Node<Key> oldlast = last;
    last = new Node<Key>();
    last.key = k; last.next = null;
    if (isEmpty()) first = last; else oldlast.next = last;
    n++;
  }
  public Key delMax() {
    if (isEmpty()) throw new NoSuchElementException("Priority queue underflow");
    Node<Key> maxptr = first;
    Node<Key> maxprev = null;
    Node<Key> ptr  = maxptr.next;
    Node<Key> prev = maxptr;
    while (ptr != null) {
      if (less(maxptr, ptr)) { maxptr  = ptr; maxprev = prev; }
      prev = ptr; ptr  = ptr.next;
    }
    Key maxitem = maxptr.key;
    if (maxprev != null) { maxprev.next = maxptr.next; }
    if (maxptr == first) first = first.next;
    if (maxptr == last)  last = maxprev;
    n--;
    return maxitem;
  }
  public boolean isEmpty() {    return n == 0; }
```

Using Unordered / Ordered Arrays

Order of growth of running time for priority queue with *n* items

Implementation	insert	delMax
Unordered array	O(1)	O(n)
Unordered list	O(1)	O(n)
Ordered array	O(n)	O(1)
Ordered list	O(n)	O(1)
Desired Gaol	*O(log n)*	*O(log n)*

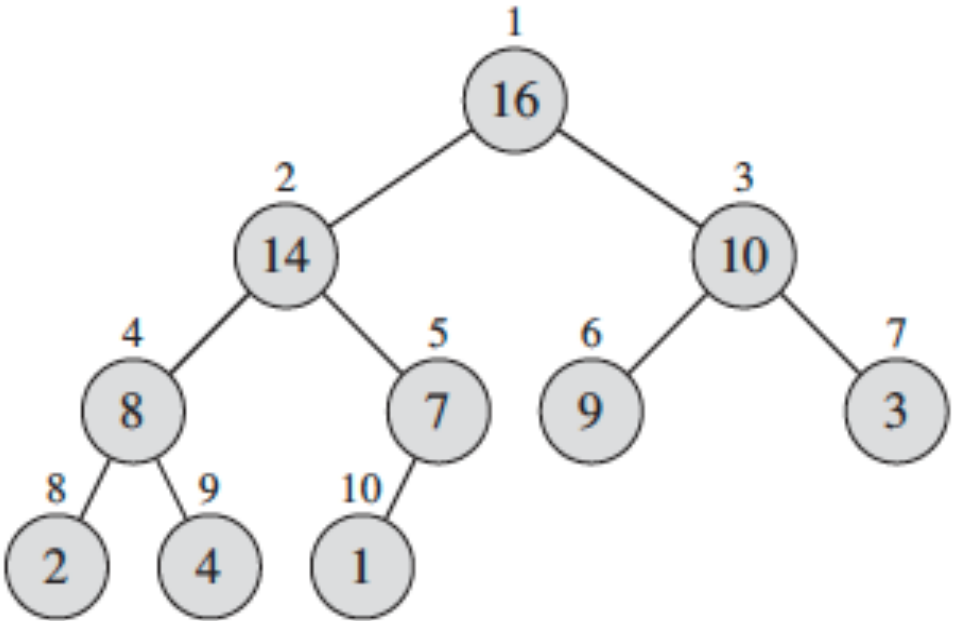
*** solution: binary heaps (partially ordered array)**

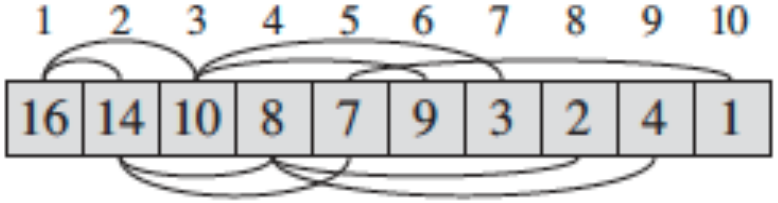
Lecture Plan

- Priority Queues API
- Elementary Implementations
- Binary Heap Data Structure
- Heap based Priority Queues
- Heapsort

Binary Heaps

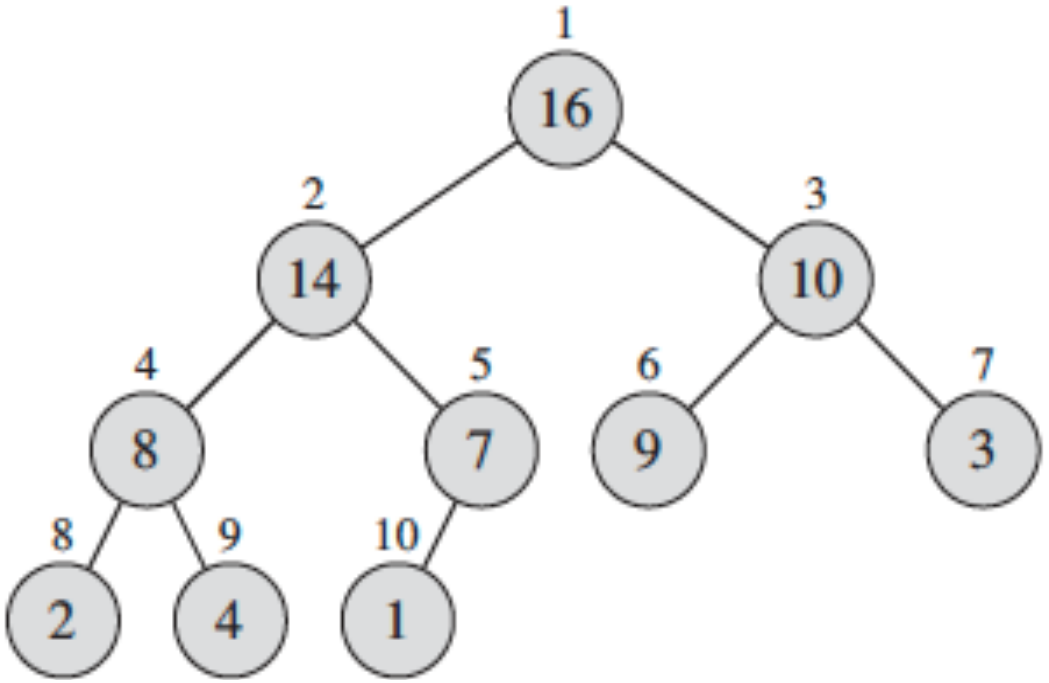
- An array object that can be visualized as a ***nearly complete binary tree***
  - the tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point





A.length = 10
A.heap_size = 10

Heaps as Trees

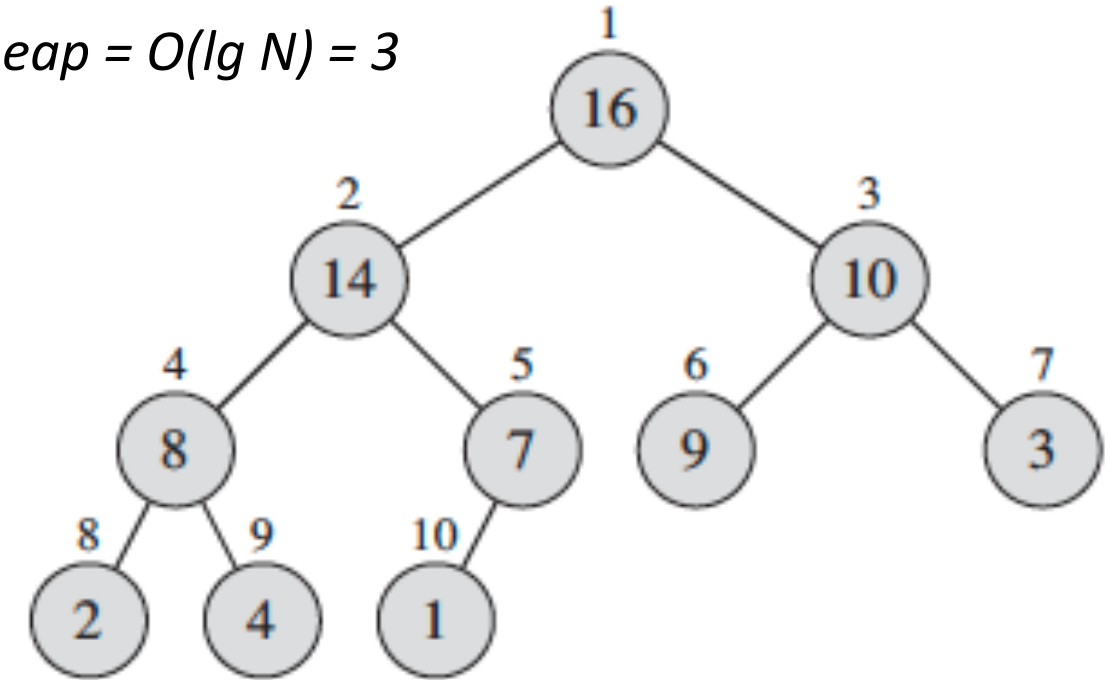



```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

PARENT(3) = floor (3 / 2) = 1
LEFT(3) = 2 * 3 = 6
RIGHT(3) = 2 * 3 + 1 = 7

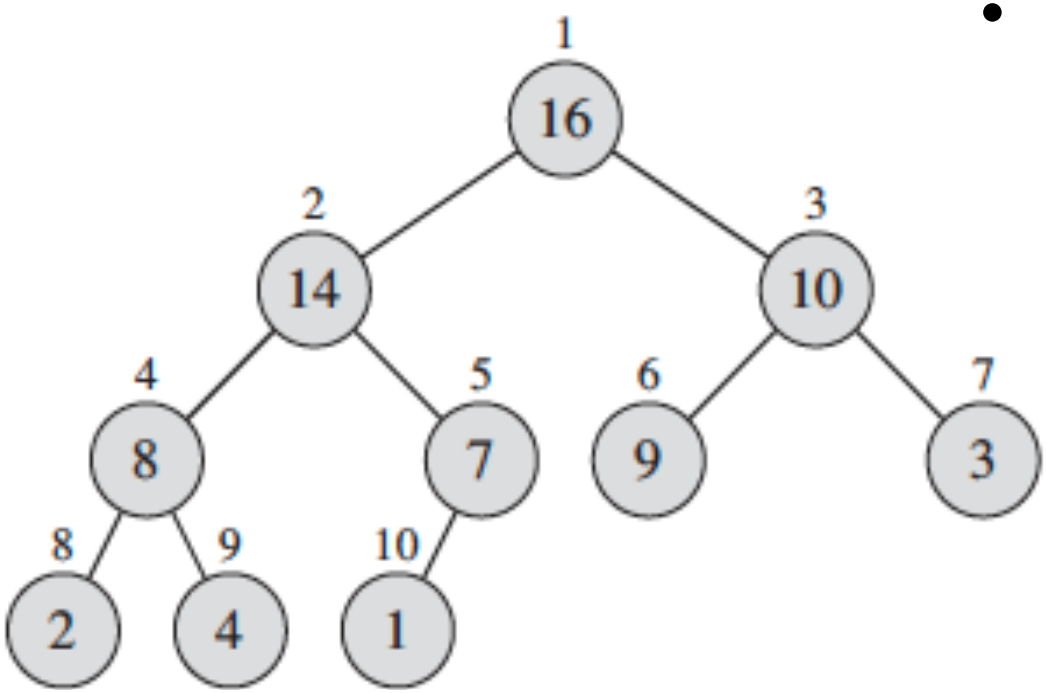
Heaps as Trees

h = height of the heap = O(lg N) = 3



- *N = number of nodes = 10*
- *number of internal nodes = floor(N/2)*
- *Height of a node = the number of edges on the longest simple downward path from the node to a leaf*

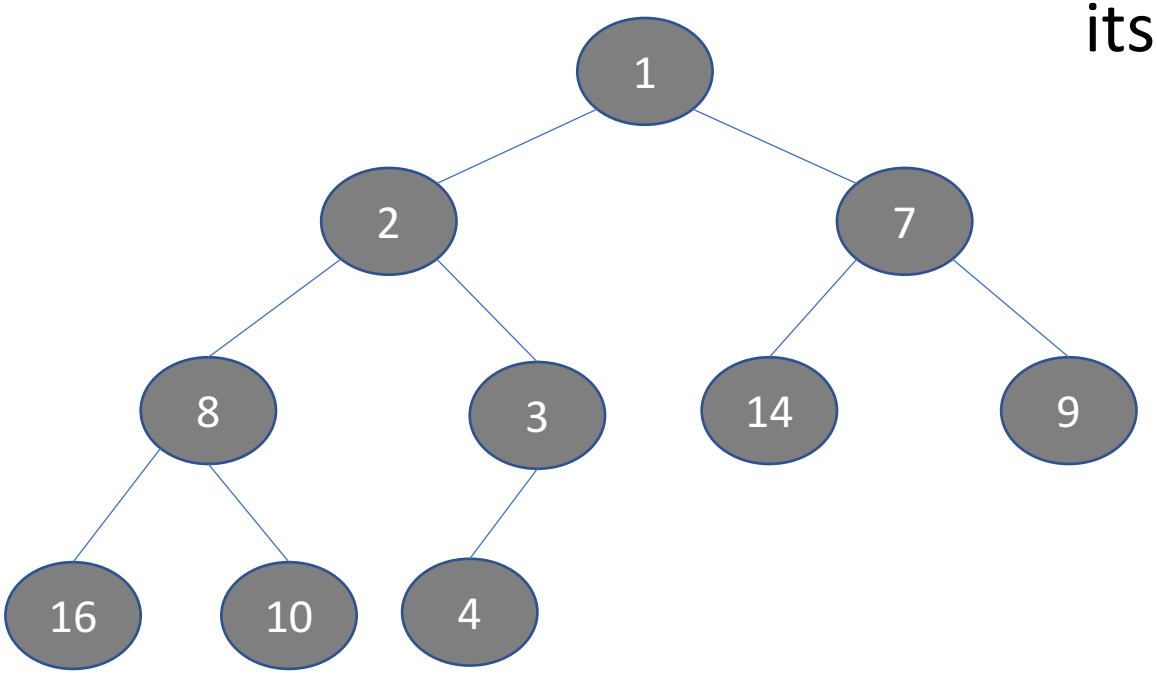
Max-Heap Property



- The key of a node is ***greater*** *than or equal to* the keys of its children:

$$A[\text{PARENT}(i)] \geq A[i]$$

Min-Heap Property



- The key of a node is ***less*** *than or equal to* the keys of its children:

$$A[\text{PARENT}(i)] \leq A[i]$$

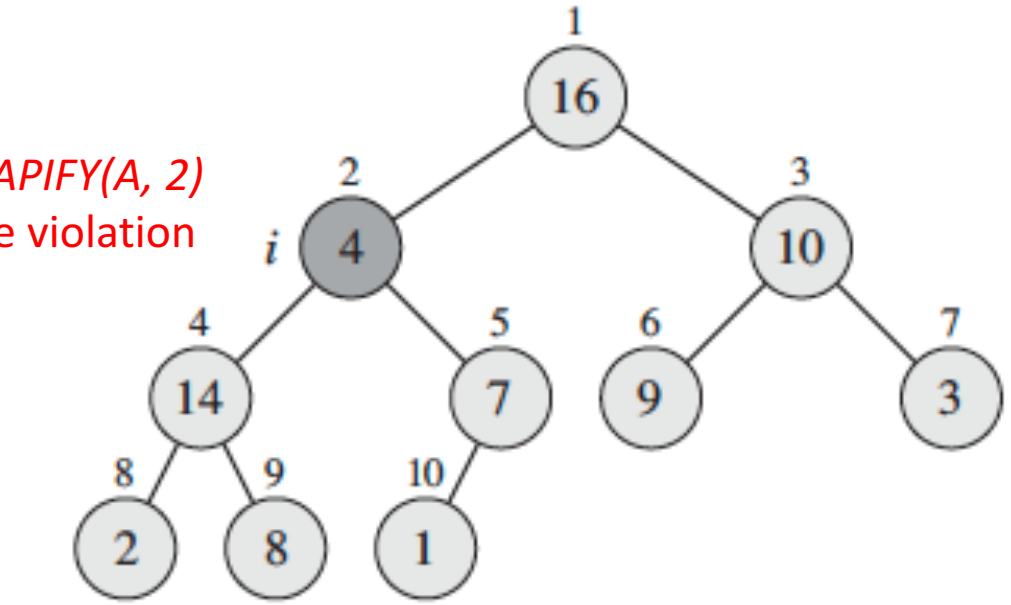
Heap Operations

- **MAX-HEAPIFY** : correct a violation of max-heap property
- **BUILD-MAX-HEAP** : build a max-heap from an unordered array
- **HEAPSORT** : use heaps to solve the sorting problem
- **MAX-HEAP-INSERT**, **HEAP-MAXIMUM**, **HEAP-EXTRACT-MAX** : use heaps to build a priority queue

MAX-HEAPIFY(A, i)

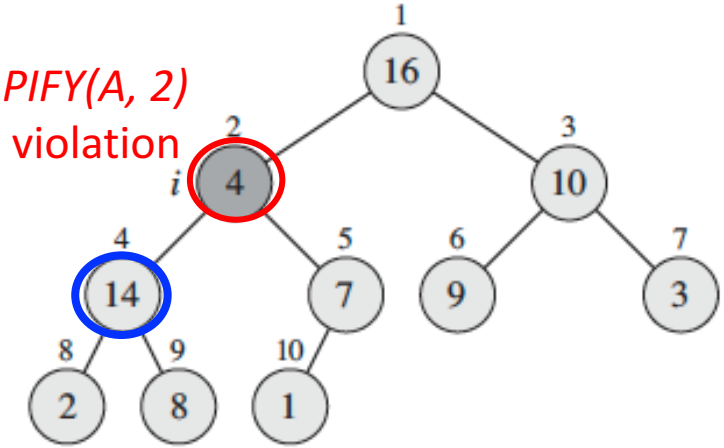
- Correct a **single** violation of the heap property in a subtree rooted at i.
- Assume that the LEFT*(i)* and RIGHT*(i)* subtrees are max-heaps
  - If A[i] violates the max-heap property, then correct the violation by "sinking" element A[i] down the tree

Call *MAX-HEAPIFY(A, 2)* to correct the violation

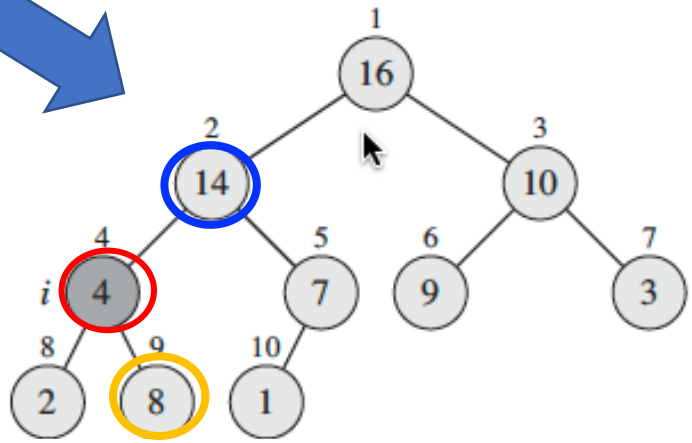


MAX-HEAPIFY(*A*, *2*)

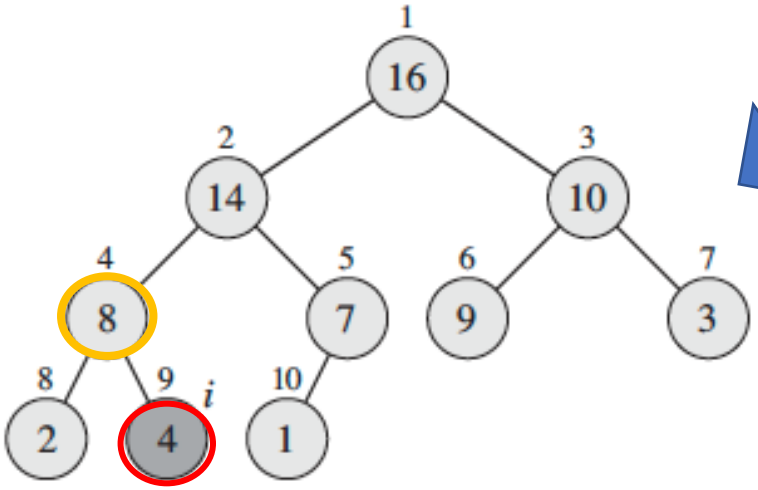
Call *MAX-HEAPIFY(A, 2)*
to correct the violation



move 4 down; move 14 up



move 4 down; move 8 up

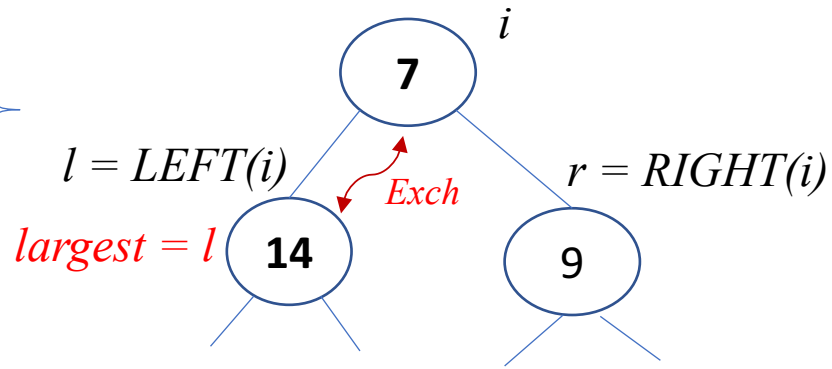


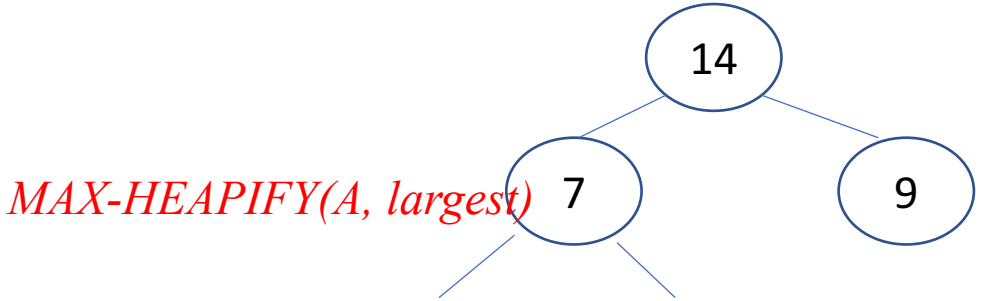
MAX-HEAPIFY: Pseudocode

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Runtime efficiency: $O(\lg N)$

Find the largest amongst three nodes:
i , *LEFT(i), RIGHT(i)*





EXERCISE.

- Illustrate the operation of MAX-HEAPIFY(A, 3) on the array

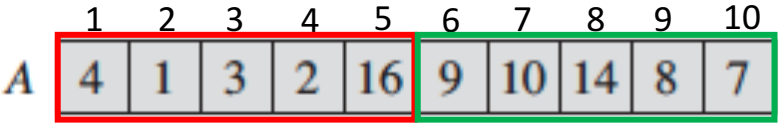
  A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]

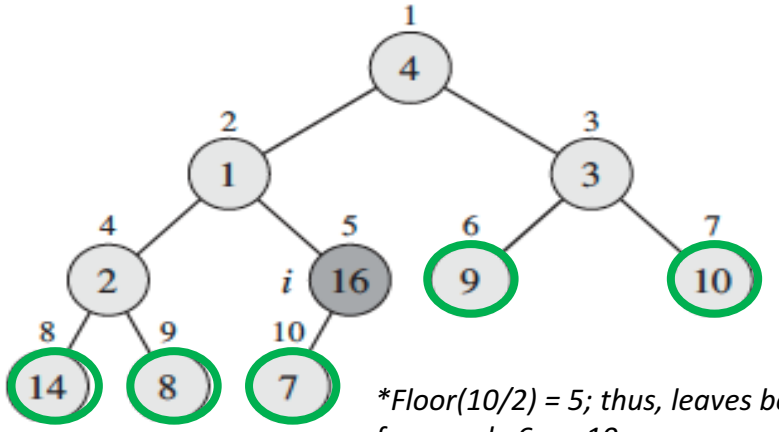
BUILD-MAX-HEAP(A)

- Converts A[1…N] to a max-heap by MAX-HEAPIFY-ing all internal nodes

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Runtime efficiency: $O(N)$





*Floor(10/2) = 5; thus, leaves begins from node 6, …, 10

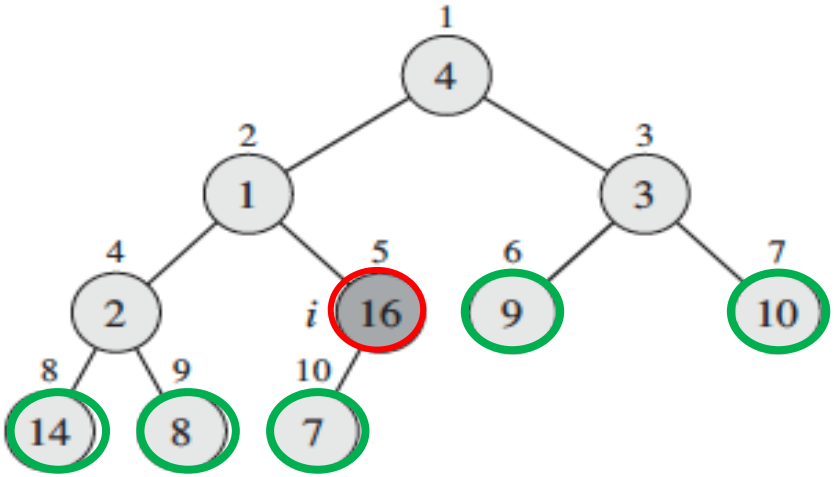
* Observe that Max-Heapify takes O(1) for time for nodes that are one level above the leaves, and in general, O(l) for the nodes that are l levels above the leaves. We have n/4 nodes with level 1, n/8 with level 2, and so on till we have one root node that is *lg n* levels above the leaves.

All leaf nodes are already a heap! (so, we don't need to apply MAX-HEAPIFY to them.

- *Leaves are the nodes indexed by*

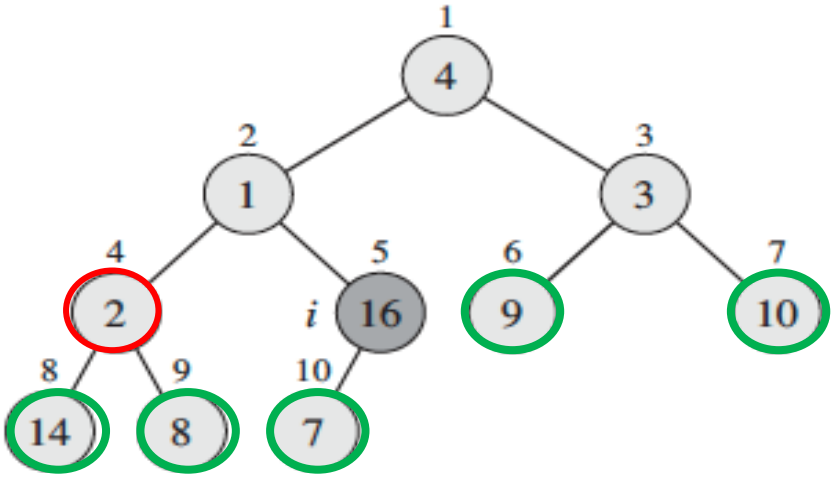
$$\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n.$$

BUILD-MAX-HEAP(A) – Demo



	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

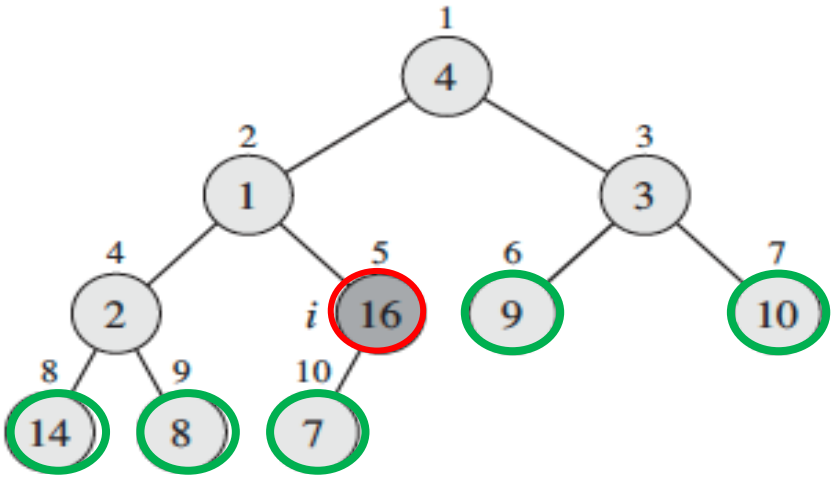
MAX-HEAPIFY(A,5)

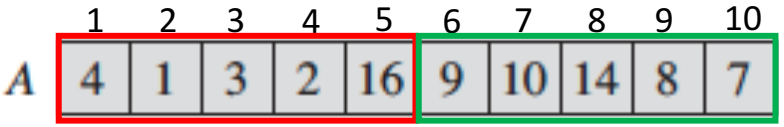


	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

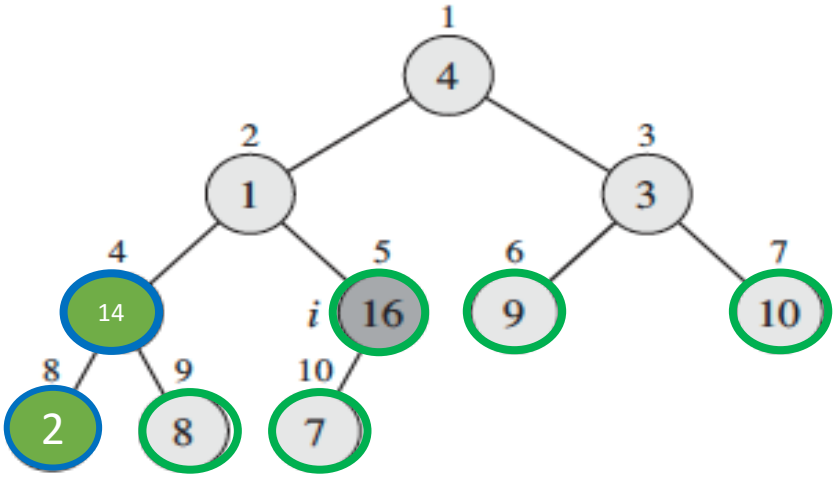
MAX-HEAPIFY(A,4)

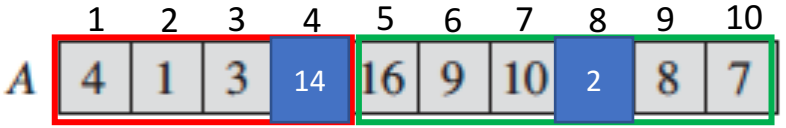
BUILD-MAX-HEAP(A) – Demo





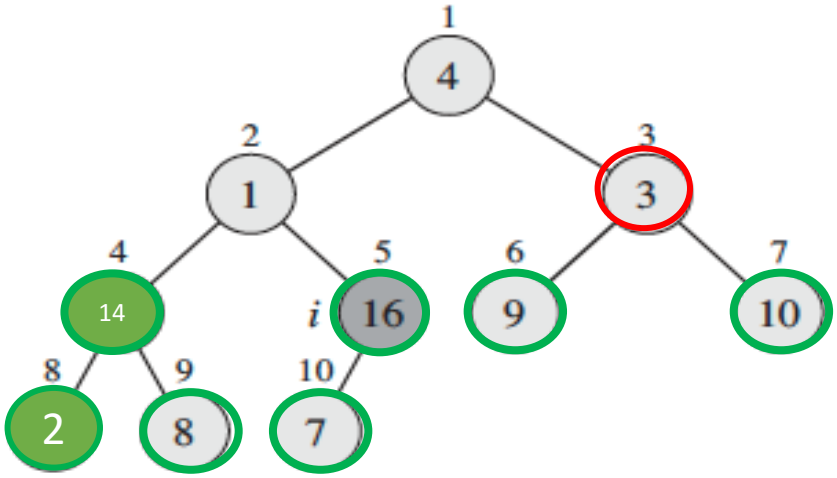
MAX-HEAPIFY(A,5)





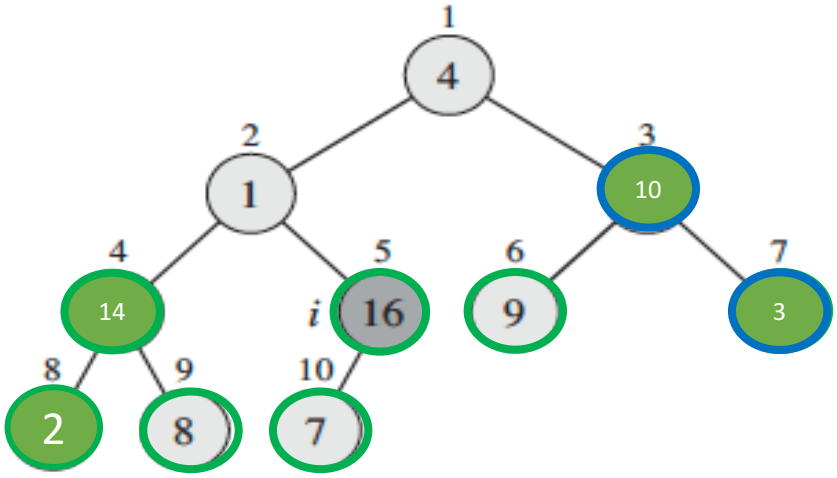
MAX-HEAPIFY(A,4)

BUILD-MAX-HEAP(A) – Demo



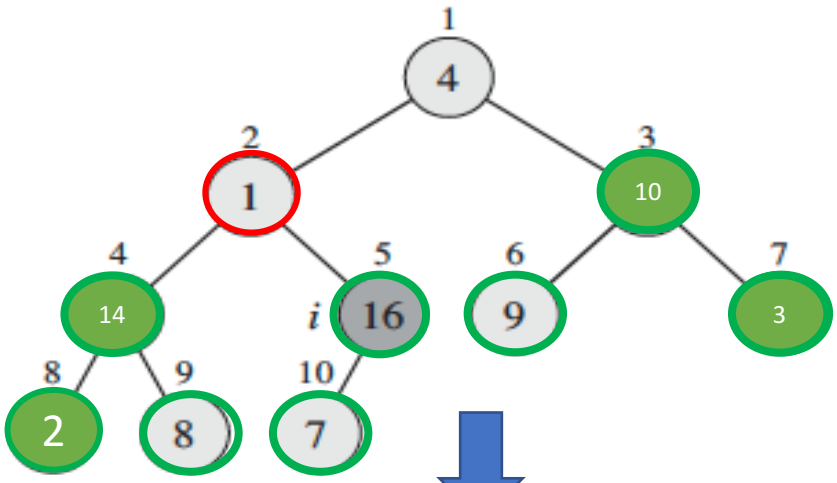
	1	2	3	4	5	6	7	8	9	10
A	4	1	3	14	16	9	10	2	8	7

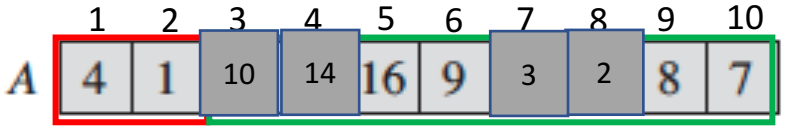
MAX-HEAPIFY(A,3)



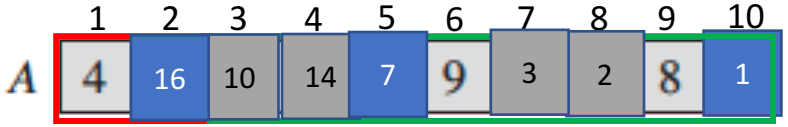
	1	2	3	4	5	6	7	8	9	10
A	4	1	10	14	16	9	3	2	8	7

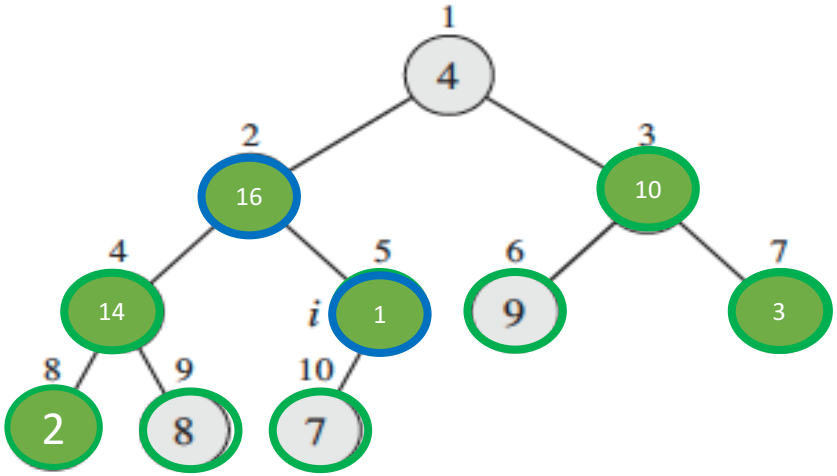
BUILD-MAX-HEAP(A) – Demo

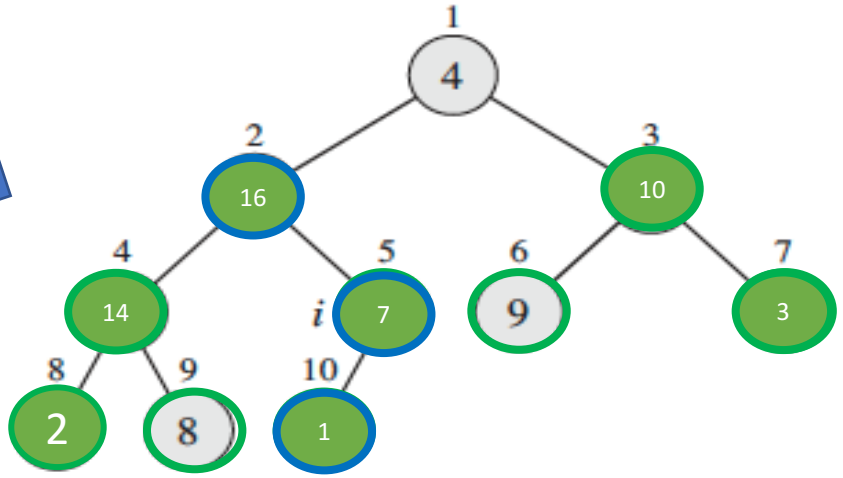




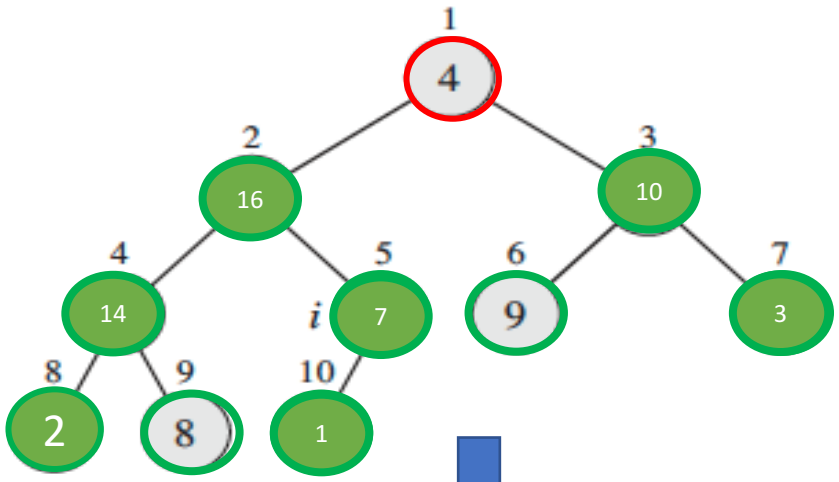
MAX-HEAPIFY(A,2)







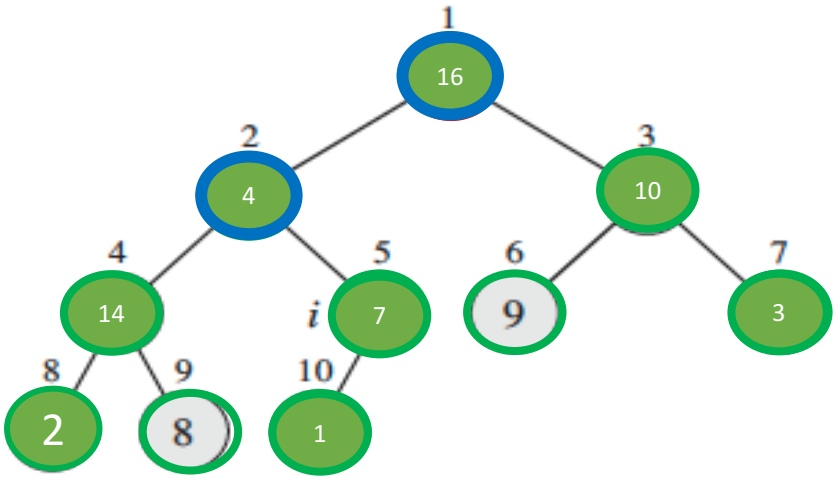
BUILD-MAX-HEAP(A) – Demo

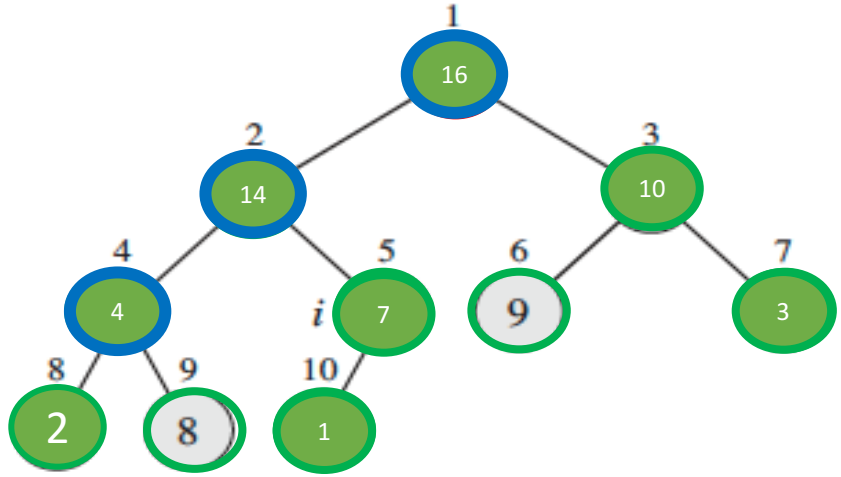


	1	2	3	4	5	6	7	8	9	10
A	4	16	10	14	7	9	3	2	8	1

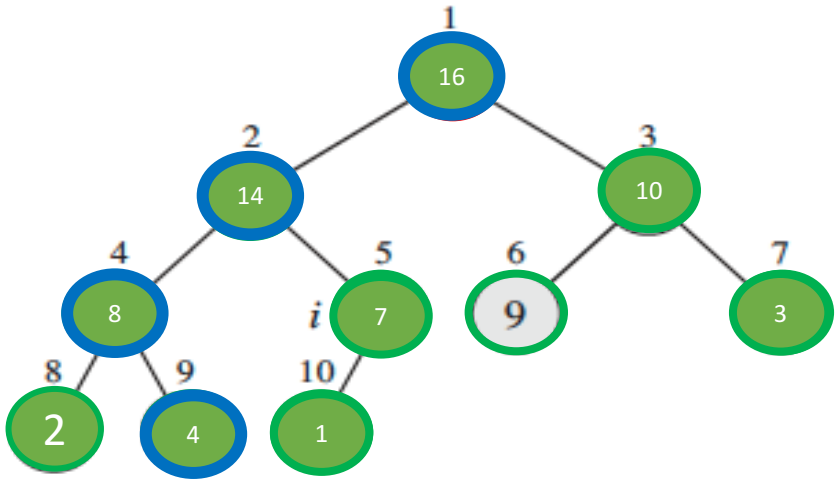
MAX-HEAPIFY(A,1)

	1	2	3	4	5	6	7	8	9	10
A	16	14	10	4	7	9	3	2	8	1

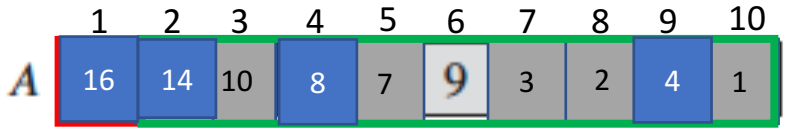




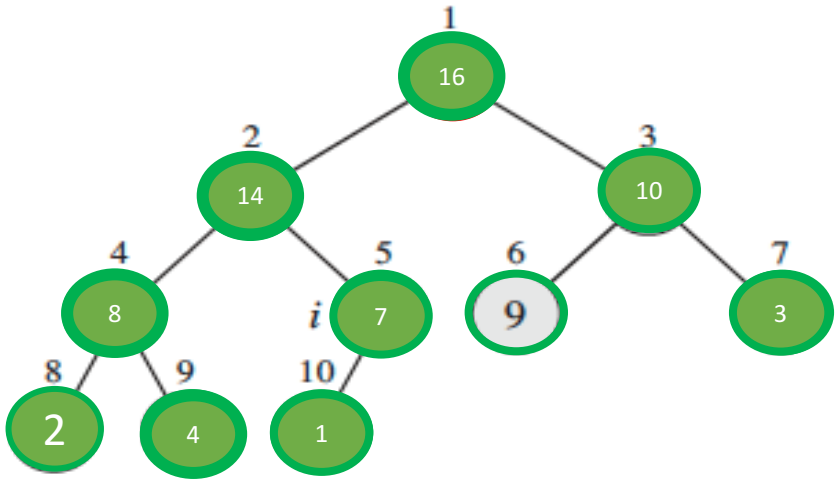
BUILD-MAX-HEAP(A) – Demo

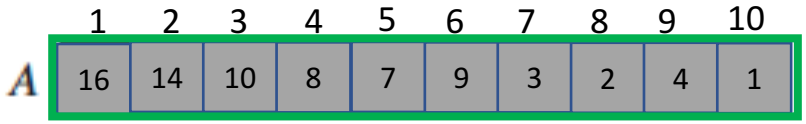


MAX-HEAPIFY(A,1)



BUILD-MAX-HEAP(A) – Demo





A.heap-size = 10
A.length = 10

EXERCISE.

- Illustrate the operation of BUILD-MAXHEAP on the array

  A = [5, 3,17,10, 84,19,6,22,9]

Lecture Plan

- Priority Queues API
- Elementary Implementations
- Binary Heap Data Structure
- Heap based Priority Queues
- Heapsort

Priority Queue

- A priority queue is an ADT for maintaining a set of elements, each with an associated value called a ***key***. A ***max-priority queue*** has three core operations
  - ***void insert(Key k)*** **:** insert an element with key=*k* into the priority queue
  - ***Key delMax()*** : remove and return the largest key

- We can use heaps to implement a priority queue.

Priority Queue

- We can use heaps to implement a priority queue as follows
  - insert
    - add the new element to the end of the heap
    - if the max-heap property is violated, lift the new element to its correct position
  - delMax
    - Remove the largest element (*i.e. the first element in the array)* from the priority queue
    - Replace the first element with the last element in the array
    - Decrease the heap-size by one
    - Call MAX-HEAPIFY(A, 1)

Heap-based Priority Queue

// delMax

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size - 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

// insert

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2        error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5        exchange A[i] with A[PARENT(i)]
6        i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = -∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Lecture Plan

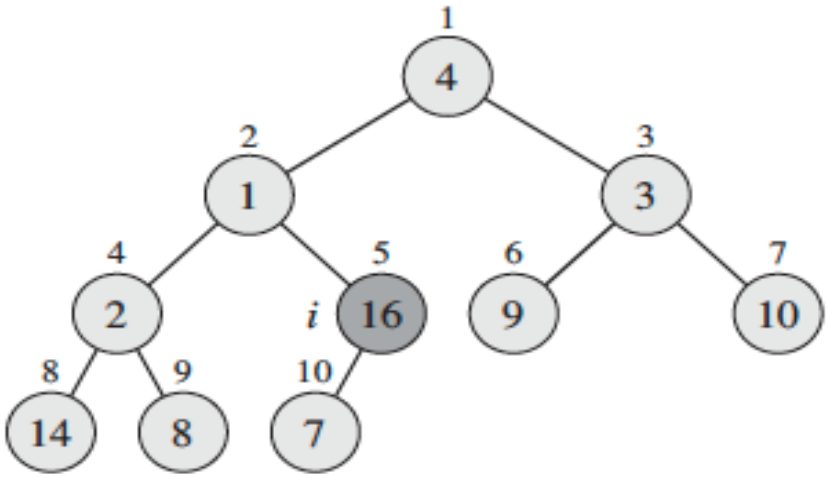
- Priority Queues API
- Elementary Implementations
- Binary Heap Data Structure
- Heap based Priority Queues
- Heapsort

HEAPSORT(A)

1. Build max-heap from unordered array *A*
   - call *BUILD-MAX-HEAP(A)*
   - Now, the maximum element is at A[1]
2. Exchange elements A[1] with A[A.heap-size]
   - Now, the maximum element is at A[A.heap-size]
3. Discard the node at A[A.heap-size] from the heap
   - By decreasing heap-size : A.heap-size = A.heap-size – 1
4. The new root A[1] may violate the max-heap property, but its left and right subtrees are max-heaps. Fix it by running MAX-HEAPIFY on the root
5. if A.heap-size != 0: Go to step 2

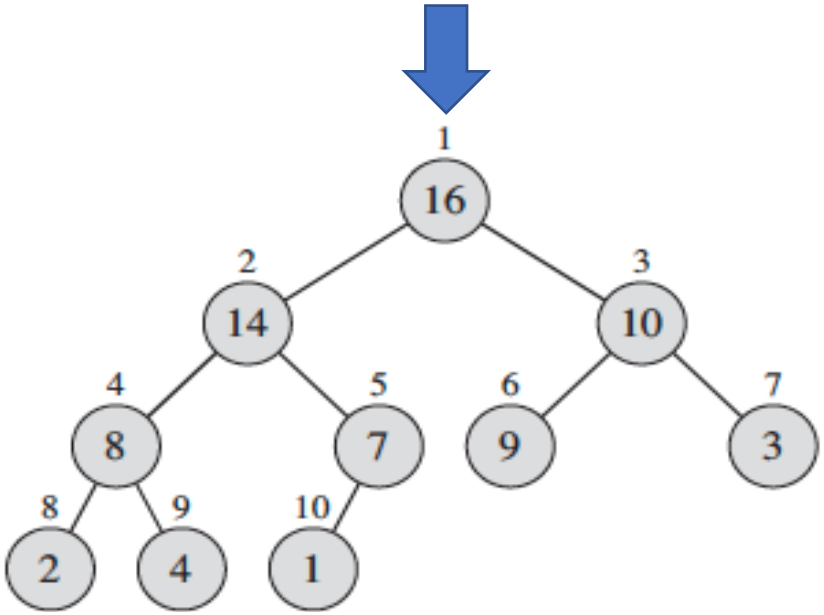
HEAPSORT(A) – Demo

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



BUILD-MAX-HEAP(A)

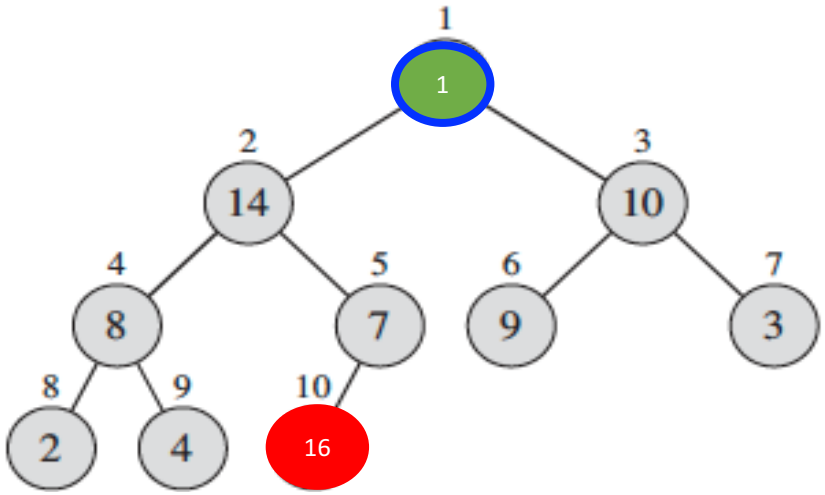
	1	2	3	4	5	6	7	8	9	10
A	16	14	10	8	7	9	3	2	4	1



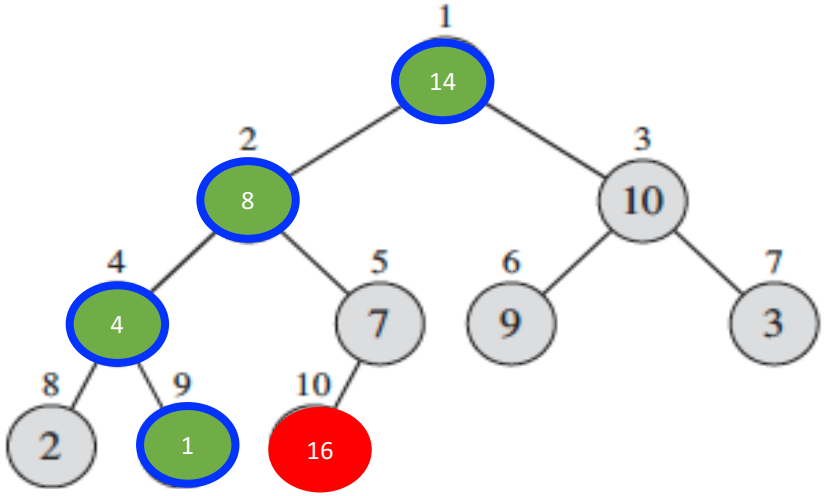
HEAPSORT(A) – Demo

A[1] ⇔ A.heap-size

A.heap-size = 10 - 1 = 9



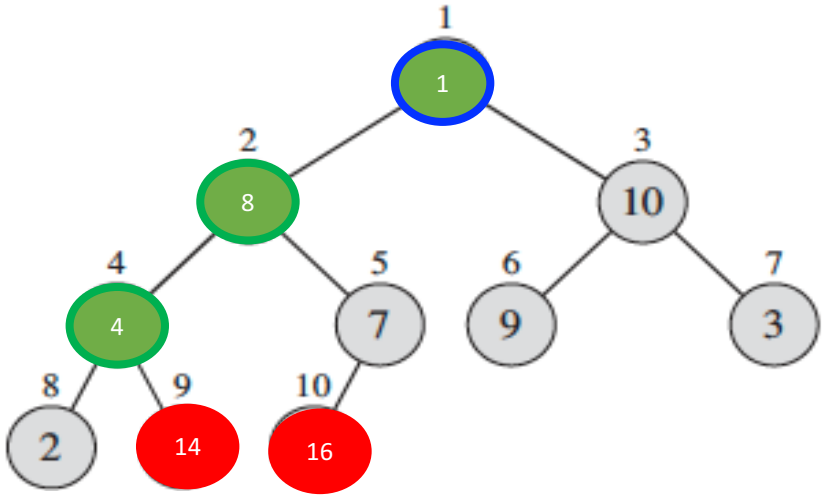
MAX-HEAPIFY(A,1)



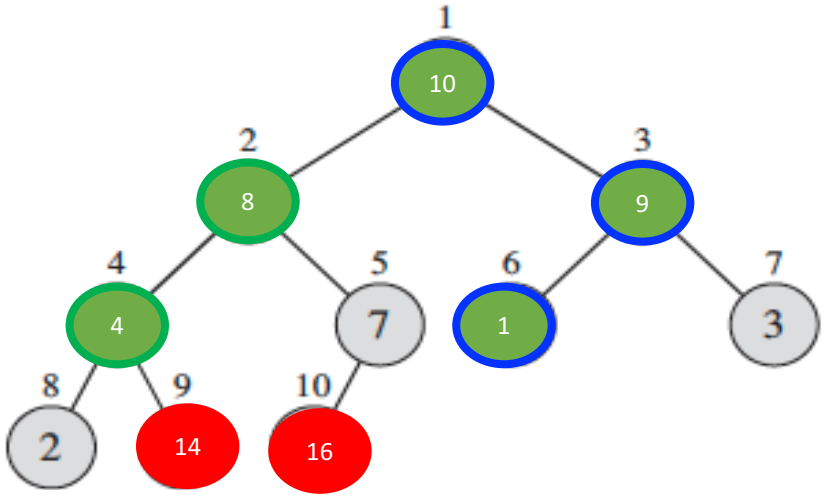
HEAPSORT(A) – Demo

A[1] ⇔ A.heap-size

A.heap-size = 9 - 1 = 8



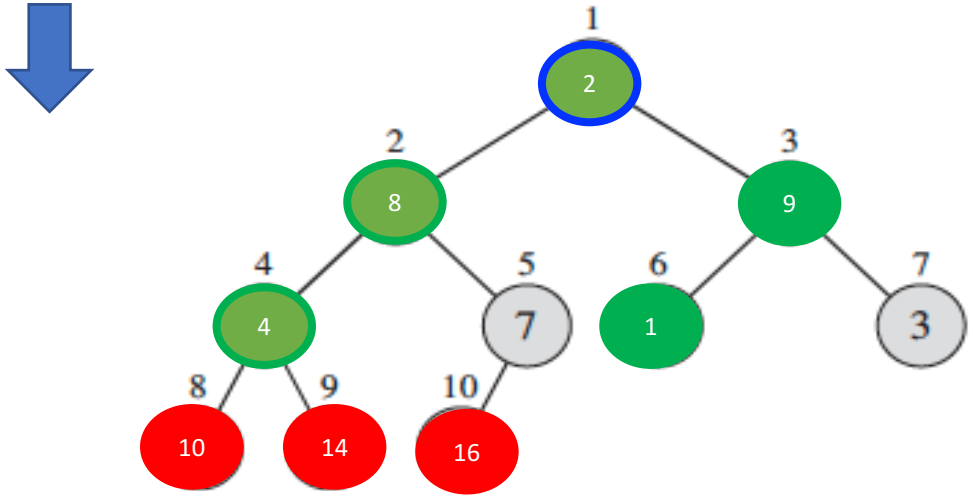
MAX-HEAPIFY(A,1)



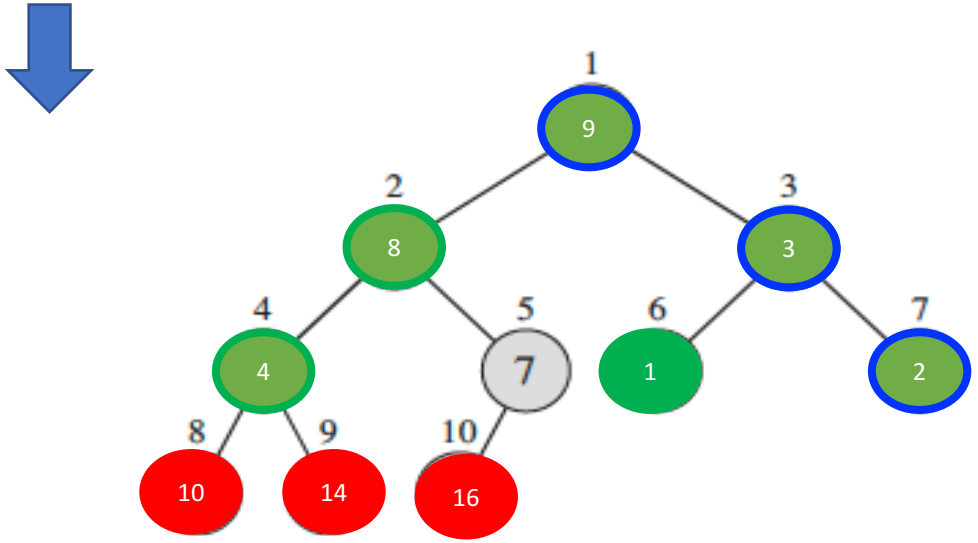
HEAPSORT(A) – Demo

A[1] ⇔ A.heap-size

A.heap-size = 8 - 1 = 7



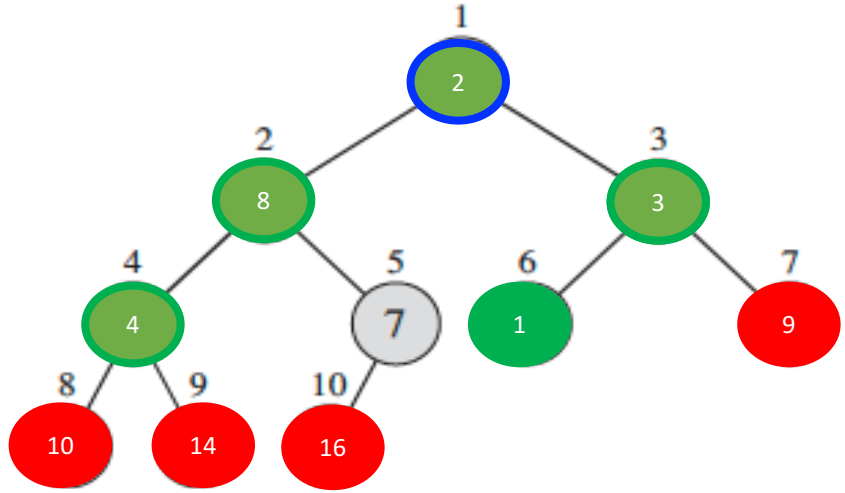
MAX-HEAPIFY(A,1)



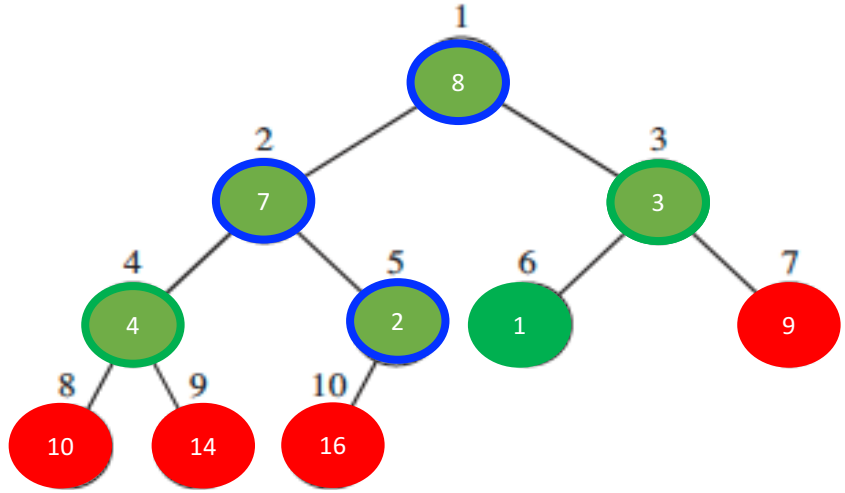
HEAPSORT(A) – Demo

A[1] ⇔ A.heap-size

A.heap-size = 7 - 1 = 6



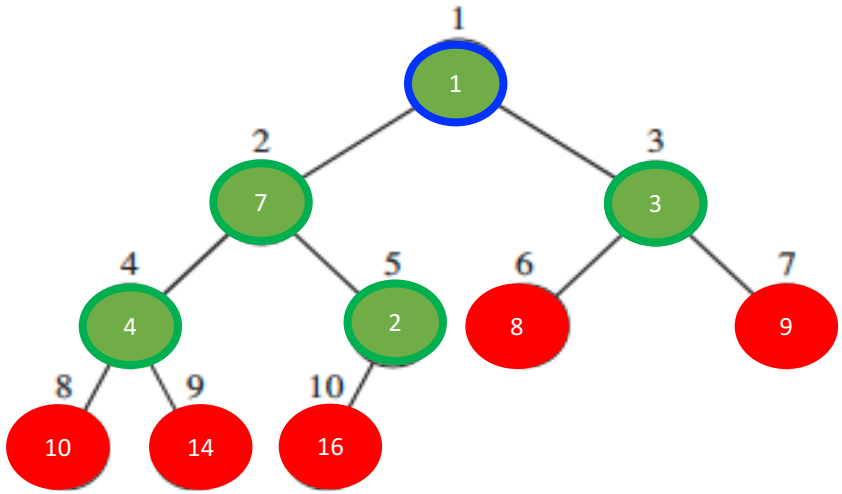
MAX-HEAPIFY(A,1)



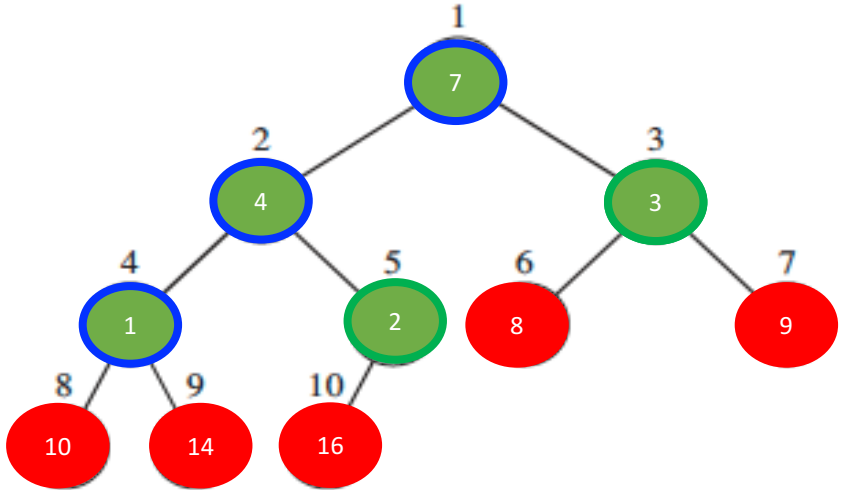
HEAPSORT(A) – Demo

A[1] ⇔ A.heap-size

A.heap-size = 6 - 1 = 5



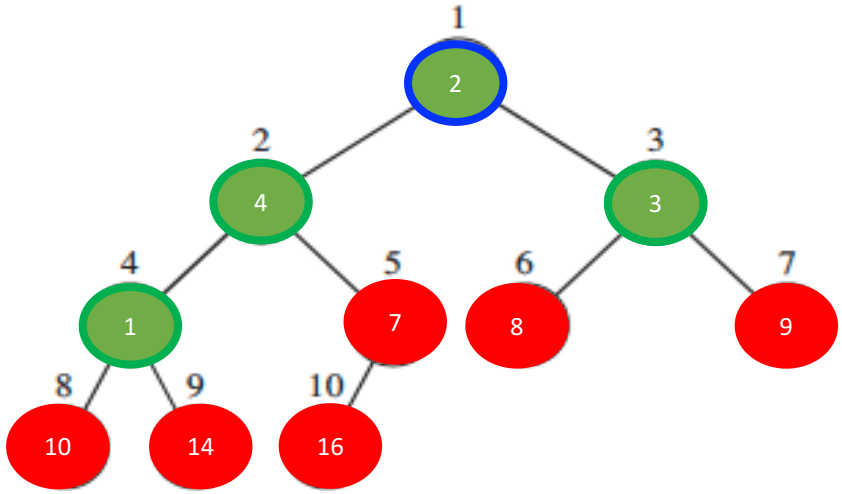
MAX-HEAPIFY(A,1)



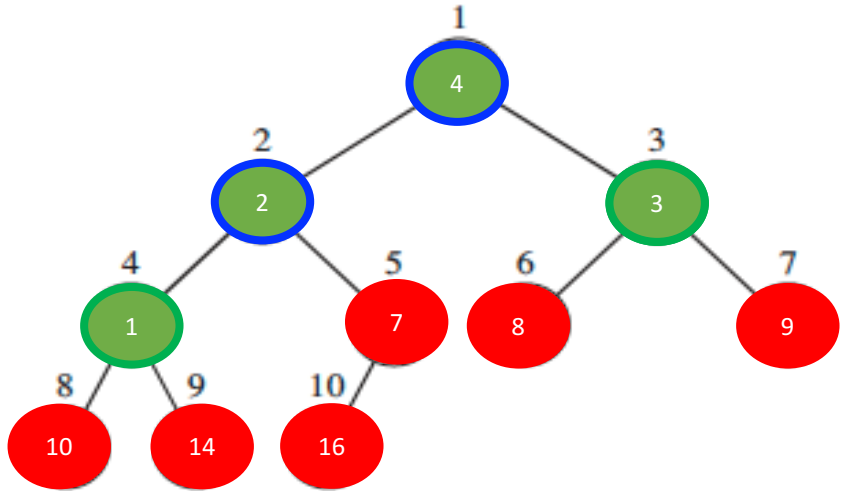
HEAPSORT(A) – Demo

A[1] ⇔ A.heap-size

A.heap-size = 5 - 1 = 4



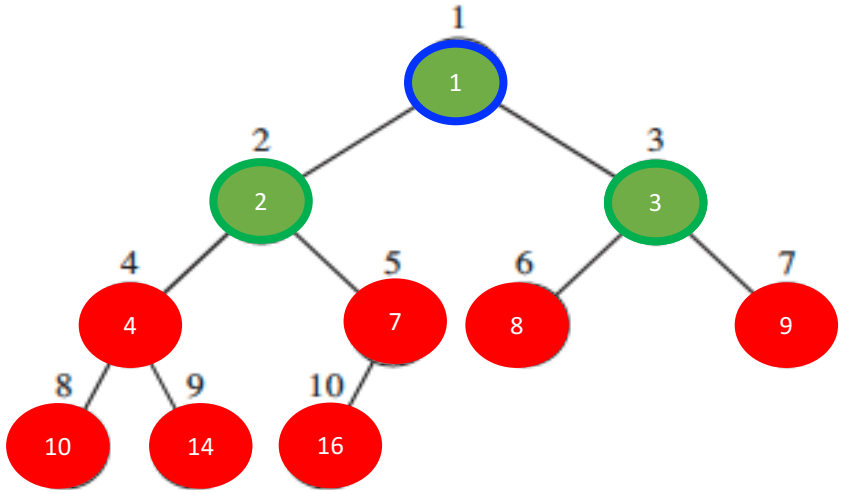
MAX-HEAPIFY(A,1)



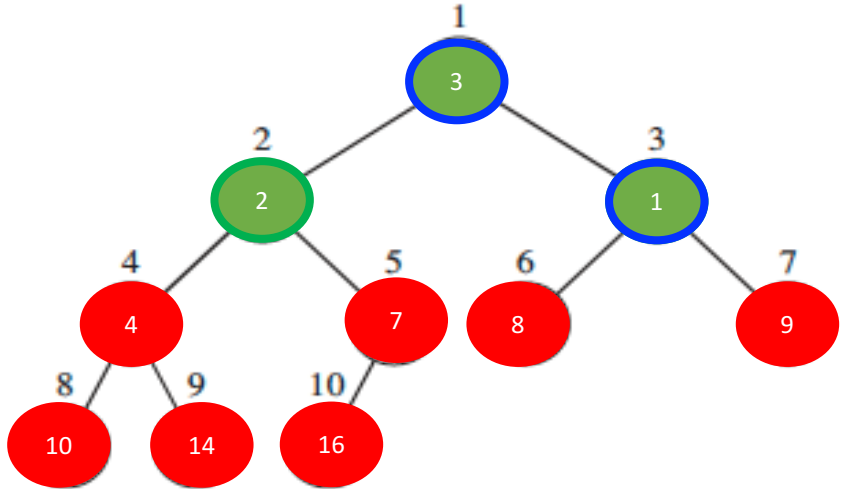
HEAPSORT(A) – Demo

A[1] ⇔ A.heap-size

A.heap-size = 4 - 1 = 3



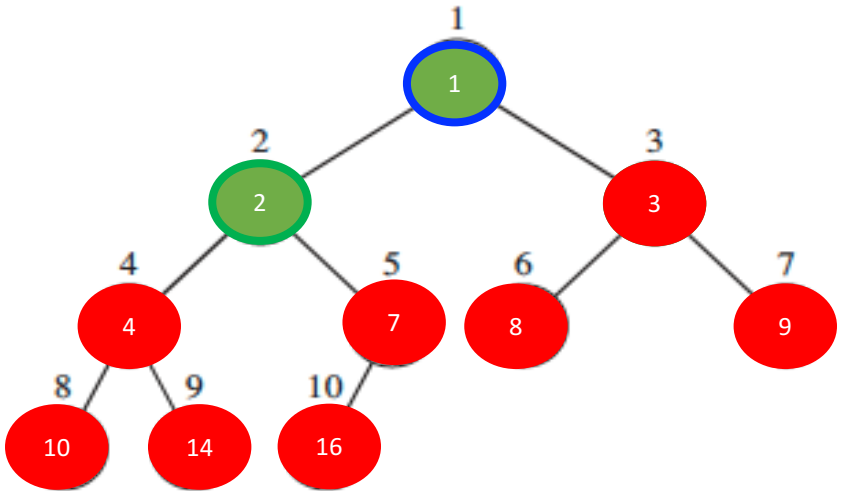
MAX-HEAPIFY(A,1)



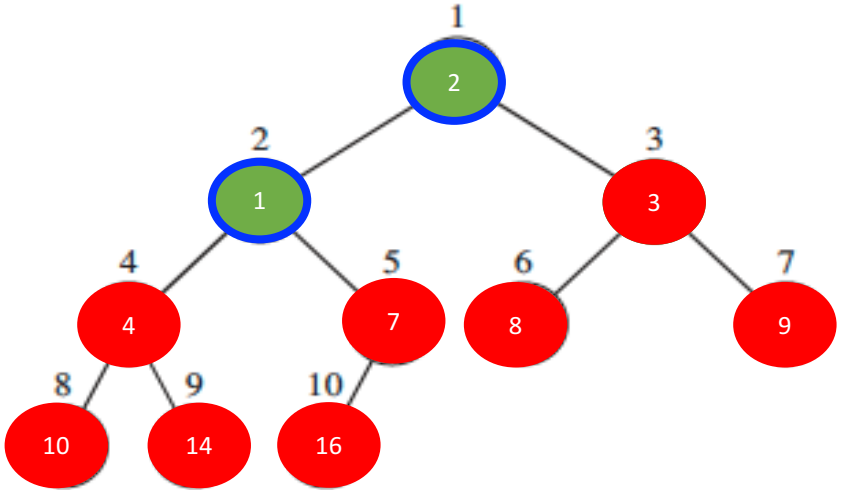
HEAPSORT(A) – Demo

A[1] ⇔ A.heap-size

A.heap-size = 3 - 1 = 2



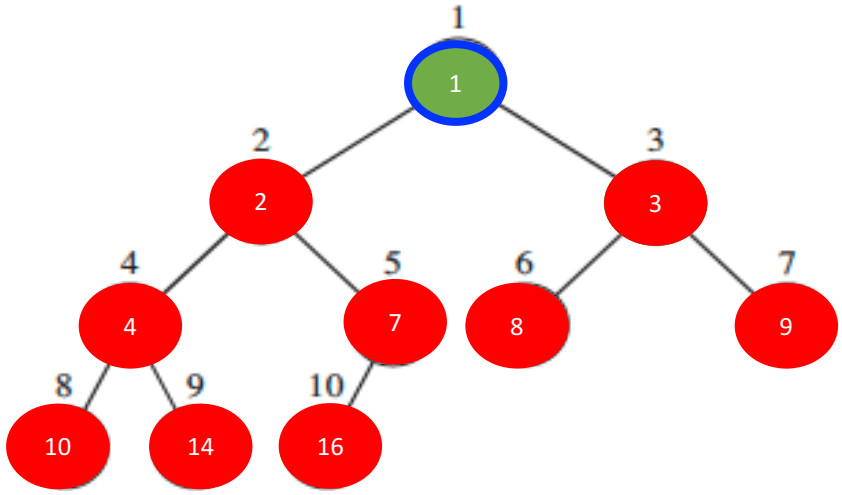
MAX-HEAPIFY(A,1)



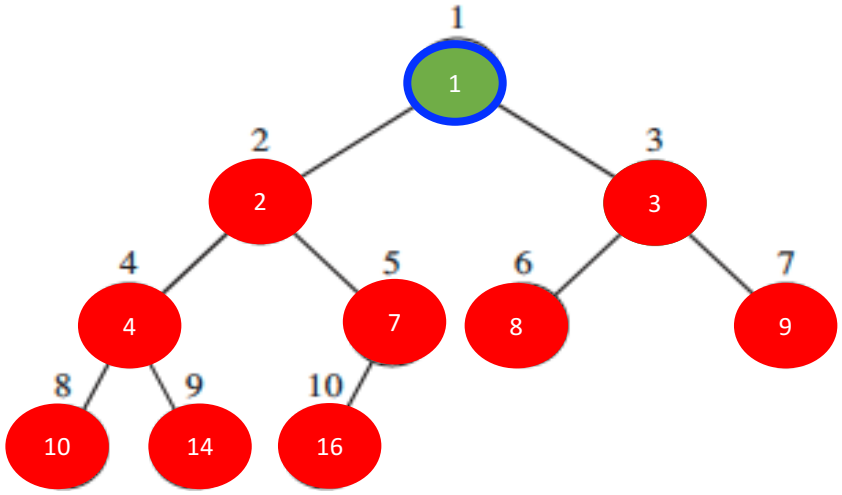
HEAPSORT(A) – Demo

A[1] ⇔ A.heap-size

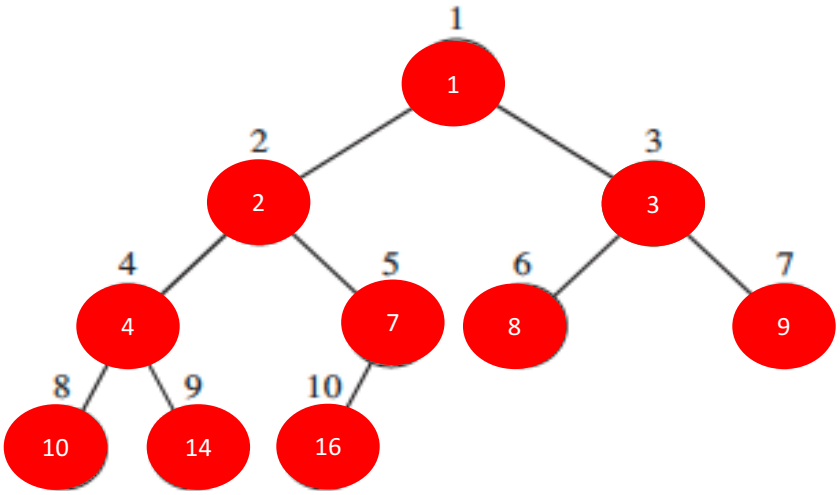
A.heap-size = 2 - 1 = 1



MAX-HEAPIFY(A,1)



HEAPSORT(A) – Demo



1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16

HEAPSORT(A) – Summary

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)                          ..... O(N)
2  for i = A.length downto 2                  ..... N-1 iterations
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)                      ..... O(lg N)
```

Runtime efficiency: $O(N \lg N)$

EXERCISE.

- Illustrate the operation of HEAPSORT on the array

  A = [5, 13,2,25, 7,17,20,8,4]